[SYSTEM-DESIGN]

Designing Data-Intensive Applications – Chapter 1: Reliable, Scalable, and Maintainable Applications

Posted by CHARLES *on* 2020-04-02

《Designing Data-Intensive Applications: The Big Ideas Behind Reliable, Scalable, and Maintainable Systems》https://amzn.to/2WYphy6

Reliability

- Continuing to work correctly, even when things go wrong.
  - aka. *fault-tolerant or resilient*.
- **Fault vs. Failure** (fault means one component failed), design system that able tolerance the fault to prevent failure;
- **Hardware Faults**: Redundancy is the key;
- **Software Errors**: could be more troublesome;
  - Developers need to make assumptions and interactions carefully.
- **Human Errors**: Design to minimize chances of errors; decouple; Test thoroughly; Quick recover ability; monitoring(performance/error rate), aka Telemetry; Good management practice/training;

Scalability:

- A system's ability to cope with increased load.
- **Describing Load**: defined by "load parameters", it depend on architecture of your system, e.g.
  - the **requests per second** to a web server,
  - the **ratio of reads to writes** in a database,
  - the **number of simultaneously active users** in a chatroom,
  - the **hit rate** on a **cache**, etc.
- Twitter hybrid approach: fanned-out vs. fetched

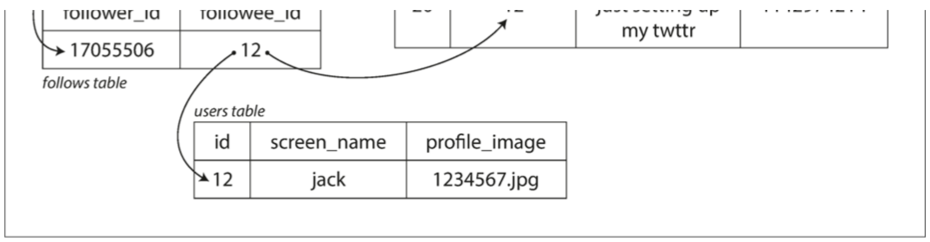

Figure 1-2. Simple relational schema for implementing a Twitter home timeline.

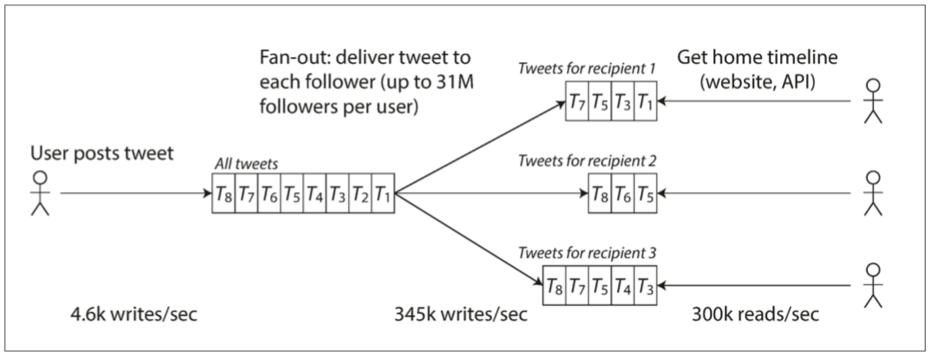

Figure 1-3. Twitter's data pipeline for delivering tweets to followers, with load parameters as of November 2012 [16].

- the average rate of published tweets is almost two orders of magnitude lower than the rate of home timeline reads → Fanned-Out approach
  - On average, a tweet is delivered to about 75 followers, so 4.6k tweets per second become 345k writes per second to the home timeline caches.
  - Moved to Hybrid approach to handle Justin-Biber effects.
- **Describing Performance**: When you increase a load parameter, keep the same system resources, what will happen ? How much do you need to increase resources to keep up the performance ?
  - E.g. "**response time**"— the time between a client sending a request and receiving a response.
  - **Latency vs. response time**: Latency is the duration that a request is waiting to be handled—during which it is latent, awaiting service.
  - Better use "**percentiles**" to measure response time, e.g. median(p50)

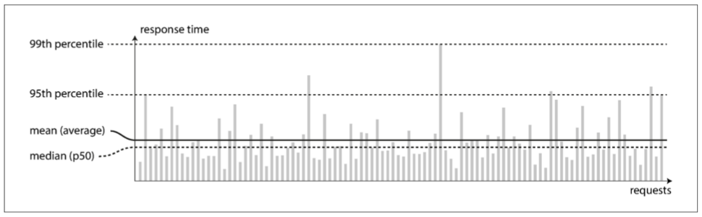

Figure 1-4. Illustrating mean and percentiles: response times for a sample of 100 requests to a service.

  - High percentiles of response times, also known as **tail latencies**, are important because they directly affect users' experience of the service.
  - Amazon describes response time requirements for internal services in terms of the 99.9th percentile (affects 1 in 1,000 requests).
    - On the other hand, optimizing the 99.99th percentile (the slowest 1 in 10,000 requests) was deemed too expensive and to not yield enough

customer satisfaction metric by 16%
    - **It is important to measure response times on the client side**.
- **Approaches for Coping with Load**:
  - Good architectures usually involve a pragmatic mixture of approaches: **Scaling up**(Vertical) & **Scaling out**(horizontal)
    - Scaling out: **Stateless** service is fairly straightforward compared to **Stateful** service.
  - It is conceivable that distributed data systems will become the default in the future.
  - The architecture of systems that operate at large scale is usually highly specific to the application.
  - An scalable architecture usually built from general-purpose building blocks arranged in familiar patterns.

Maintainability:

- **Operability**: Make it easy for operations teams to keep the system running smoothly.
- **Simplicity**: Make it easy for new engineers to understand the system,
  - By removing as much complexity as possible from the system. (Note: this is not the same as simplicity of the user interface.)
  - Making a system simpler DOES NOT necessarily mean reducing its functionality;
  - **Complexity**: as accidental if it is not inherent in the problem that the software solves (as seen by the users) but arises only from the implementation.
    - One of the best tools we have for removing accidental complexity is **abstraction**. (However,finding good abstractions is very hard.)
- **Evolvability**:Make it easy for engineers to make changes to the system in the future, adapting it for unanticipated use cases as requirements change. Also known as **extensibility**, **modifiability**, or **plasticity**.
  - **Agile** working patterns provide a framework for adapting to change
  - simple and easy-to-understand systems are usually easier to modify than complex ones (Linked to the previous point – **Maintainability**)

Summary

- **functional requirements**: (what it should do, such as allowing data to be stored, retrieved, searched, and processed in various ways),
- **nonfunctional requirements**: (general properties like security, reliability, compliance, scalability, compatibility, and maintainability)
- **Reliability** means making systems work correctly, even when faults occur.
- **Scalability** means having strategies for keeping performance good, even when load increases.